Índex

1 INTRODUCCIÓ

Aquesta memòria conté el treball de final de carrera que hem desenvolupat en el grup de recerca en Intel·ligència Artificial de la Universitat de Lleida l'objectiu del qual és el disseny i implementació d'un sistema de traducció de format textual a format gràfic per a un model recursiu de còmput d'arguments garantits basat en la semàntica formal RP-DeLP (Recursive Possibilistic Logic Programming) desenvolupat per Alsinet-Béjar-Godo, 2010.

Per complir aquest requisit es dissenyarà i implementarà un traductor en Java. Un **traductor** és un programa que transforma un text o un programa escrit en llenguatge font en un altre text o programa equivalent escrit en un altre llenguatge produint, si cal, missatges d'error. Per tant, el TFC consistirà en desenvolupar un traductor en el que se li introduirà un text de produccions del model recursiu de còmput d'arguments garantits i s'obtindrà la representació gràfica d'aquest tal i com veiem en la figura1.

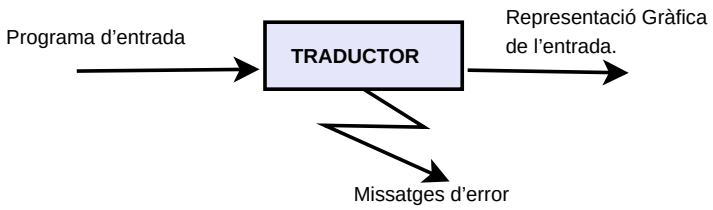

Figura 1: Representació bàsica del traductor a construir

La implementació d'aquest traductor s'ha decidit realitzar en Java per dues raons. D'una banda la disponibilitat d'eines de suport al disseny de traductors en Java i, per una altra, facilitar una posterior adaptació al Javascript per poder-lo utilitzar en una pàgina web.

La memòria s'estructura de la forma següent:

- Presentació del llenguatge d'entrada que accepta el traductor.
- Disseny general del traductor.
- Descripció de l'anàlisi lèxic.
- Descripció de l'anàlisi sintàctic.
- Traducció dirigida per la sintaxi i la taula de símbols.
- Explicació de la representació gràfica.
- Conclusions.

2 DESCRIPCIÓ DEL LLENGUATGE D'ENTRADA

2.1 Produccions

El traductor/compilador acceptarà una o més produccions com a entrada que seran del tipus següent:

$$(Variable)\{Garantia\}\,;$$

La part anomenada **Variable** estarà delimitada pels símbols (). Aquesta part conté el nom de la variable així com el seu grau separats per una coma. Els noms del les variables seran de la forma: [A-Za-z]([A-Za-z]|[0-9]|_)*. Aquestes variables es poden negar al posar davant el símbol "¬". El grau estarà en l'interval $[0-1]$sobre $\Re$. El símbol per delimitar els decimals serà el punt.

La part anomenada **Garantia** estarà delimitada pels símbols {}. Aquesta part apareixeran les garanties de la variable de la primera part de la producció. Quan la variable tingui grau 1 aquesta part només contindrà els delimitadors {}. Si el grau és diferent a 1 tindrem la variable que apareix en la primera part de la producció seguit del símbol d'implicació format per "<"més "-" i a continuació una o més variables separades pel símbol de la conjunció "^". Aquestes variables tindran la mateixa forma que que la del apartat anterior.

Per poder fer el tractament d'errors en el programa tota producció acabarà amb el símbol ";".

Les diferents produccions de l'entrada han d'estar ordenades per ordre descendent del grau de la variable que apareix en la primera part de la producció.

2.2 Comentaris

El programa també admet comentaris de dos tipus:

El símbol **//** al principi d'una línia farà que tot el que s'escrigui després en la mateixa línia sigui eliminat per l'analitzador lèxic.

Els símbols **/* */** . Tot el que s'escrigui entre aquest símbols també serà eliminat per l'analitzador lèxic.

2.3 Exemple

Un exemple de llenguatge d'entrada que processarà el traductor és el següent:

```
/*Això és un exemple*/
(P,1) {} ;
(B_1,0.9) {B_1<-P} ;
(V,0.8) {V<-P} ;
(¬S,0.7) {¬S<-P^V} ;
//EOF
```

3 DISSENY GENERAL DEL TRADUCTOR

El traductor que es dissenya és un compilador d'una sola passada perquè només es llegirà el codi font un sol cop. La gramàtica que accepta és de lliure context on hi ha declaració explícita de variables en la zona definida entre parèntesis, la zona d'argument. El llenguatge no està estructurat en blocs, es a dir, no hi ha àmbits d'utilització i visió d'algunes variables com podria ser una funció.

3.1 Esquema general

En aquest apartat s'explica, esquemàticament, les parts de les que constarà el traductor. En posteriors seccions s'explica detalladament com s'implementen cadascuna de les parts.

El traductor que es dissenya té dues fases diferenciades:

- Fase d'anàlisis: on es comprova la correcció del fitxer d'entrada i es creen les estructures necessàries per a la següent fase. Aquesta fase és capaç de generar errors.
- Fase de síntesis: on, a partir de les estructures construïdes en la fase d'anàlisis, es crea un objecte o representació gràfica.

3.1.1 Fases d'anàlisi

La fase d'anàlisi, per poder realitzar la seva feina, es divideix en tres parts més:

- Anàlisi lèxic: Té la funció de dividir l'entrada en components bàsics del llenguatge a traduir. Aquests components bàsics estan formades per seqüències de caràcters.
- Anàlisi sintàctic: comprova que els components bàsics que obtenim en la fase anterior segueixen la estructura que del llenguatge que traduïm.
- Anàlisis semàntic: s'encarrega de comprovar que el fitxer d'entrada tingui coherència en el seu significat. Es a dir, mira que els rangs de les variables siguin correctes, si les variables existeixen,...etc.

Totes aquestes fases són capaces de generar els seus propis errors en la fase d'anàlisi. Per tant, tindrem 3 tipus d'errors: lèxic, sintàctic i semàntic.

3.1.2 Taula de Símbols

Una funció dels traductors és guardar informació sobre els identificadors i els seus atributs que es troben al llarg del fitxer d'entrada. Aquesta informació es guarda en una estructura de dades que és creada en la fase d'anàlisis a mesura que es llegeix l'entrada. Tant en la fase d'anàlisi com de síntesis s'accedeix a la taula de símbols, per tant, és un element cabdal que està íntimament lligat al traductor.

3.1.3 Fase de síntesis

Un cop s'ha comprovat que el programa d'entrada és correcte mitjançant la fase d'anàlisis, utilitzant elements generats a l'etapa anterior com la taula de símbols, el traductor genera el resultat d'aquest programa d'entrada que serà una representació gràfica.

3.2 Eines emprades

Per la implementació del anàlisi lèxic en s'utilitza JFLEX[1] que és un generador d'analitzadors lèxics o scanners. S'escull aquesta eina pels següents motius:

- Genera el codi en Java
- Facilitat d'interacció amb l'eina CUP utilitzada en fases següents.

Per implementar l'anàlisi sintàctic s'utilitza el CUP[2] que és una analitzador sintàctic LALR escrit en C. Els motius de la seva elecció són:

- Genera el codi de l'analitzador en Java.
- Integració senzilla amb l'eina JFLEX emprada en la realització de l'analitzador lèxic.
- Llibertat d'elecció de l'estructura de dades que s'utilitzarà en les fases posteriors.
- Mecanisme de gestió d'errors més avançat que altres alternatives en Java, com JavaCC, que s'aturen al detectar un error.

Per últim es fa servir la llibreria JGRAPHX[3] per implementar la part gràfica del traductor degut als següents motius:

- Està implementat en java.
- I la versió d'aquest paquet permet implementar fàcilment una aplicació web per implementar grafs.

Per implementar les estructures de dades que es necessiten al llarg del programa es fa anar les estructures incloses en l'API de Java[4].

[1] www.jflex.de
[2] http://www2.cs.tum.edu/projects/cup/
[3] http://www.jgraph.com/
[4] http://docs.oracle.com/javase/7/docs/api/

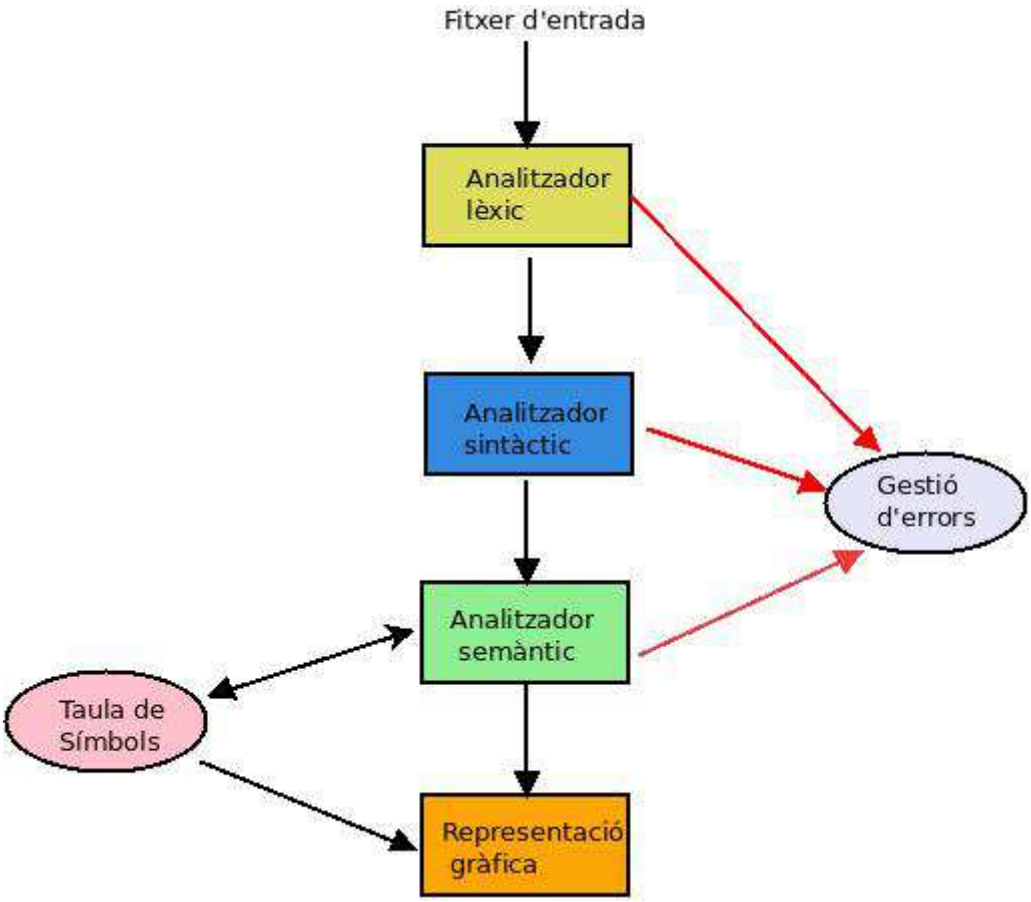

Figura 2: Esquema General del Compil·lador

4 ANÀLISI LÈXIC

4.1 Funcions

L'anàlisi lèxic és la primera fase d'anàlisi. La seva funció principal és la de llegir la cadena de caràcters de l'entrada i agrupar-los en components lèxics o tokens. Aquests tokens es passen a l'analitzador sintàctic que els utilitzarà en la següent fase d'anàlisi. Altres funcions de l'analitzador lèxic són:

- L'eliminació dels comentaris del programa.
- L'eliminació del espais en blanc, tabuladors, retorns de carro...etc.
- Detectar els errors lèxics.
- Portar el compte de quina línia és la que estem llegint.

4.2 Implementació

L'estructura de l'especificació JFLEX del projecte, com qualsevol especificació JFLEX, consta de les següents parts:

CODI D'USUARI
%%
OPCIONS I DECLARACIONS DE MACROS
%%
REGLES LÈXIQUES

on els símbols %% actuen com a separadors de secció.

4.2.1 Codi d'usuari

El contingut d'aquesta secció del programa es copia al peu de la lletra al principi de l'analitzador lèxic que es crea (abans de la declaració de la classe). En aquesta secció només hi van les instruccions *import* o *package* que s'utilitzin.

4.2.2 Opcions i declaracions

En aquest apartat es defineixen les **opcions** que permeten personalitzar l'analitzador lèxic que s'obtindrà. L'opció més importants que conté l'especificació JFLEX del projecte és *%cup*, que ens permet activar el mode de compatibilitat de JFLEX amb CUP. L'opció *%cup* equival a les clàusules JFLEX següents:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
return new java_cup.runtime.Symbol(<CUPSYM>.EOF);
%eofval}
%eofclose
```

S'especifica que es tracta d'un text amb *%unicode*, el nom que tindrà la classe que obtindrem amb *%class* i, finalment, que es guardi informació sobre la línia que tractem amb *%line*.

El codi inclòs entre *%{...%}* s'inclou al peu de la lletra dins la classe que es generarà. En el programa creem dues funcions auxiliars que creen objectes java-cup.runtime.Symbol amb informació sobre la línia, la columna i, s'hi en té, el valor del token que estem tractant.

Per últim, en aquesta secció tenim les **macros** que serveixen per donar identificadors a expressions regulars que fan més fàcil la elaboració i comprensió de les especificacions lèxiques. Els macros tenen la forma:

identificador_macro = expressió_regular

El programa es serveix dels macros per definir: les variables, el rang de les variables, els comentaris i els espais en blanc que hem explicat en l'apartat 2. Les variables les defineix com un conjunt alfanumèric format per una o més lletres seguit d'un número o un guió baix. El rang del grau en l'interval $[0-1]$ sobre $\Re$. I els comentaris són els habituals // al principi de línia i tot el text entre /* i */.

4.2.3 Regles lèxiques

L'última part de la especificació són un conjunts d'expressions regulars i accions - codi en Java- que s'executen quan el analitzador lèxic té una entrada que casa amb la expressió regular. Si hi ha més d'una expressió que coincideix amb l'entrada JFLEX selecciona aquella que coincideix amb una porció més gran. En el cas que dues expressions agafin la mateixa porció de l'entrada s'escull aquella que està escrita en primer lloc.

En el projecte s'utilitza aquest apartat per definir el tractament que fem als tokens que anem trobant durant la lectura de l'entrada. Concretament, mitjançant les funcions auxiliars creades en l'apartat anterior, passem tots els delimitadors que anem trobant al analitzador sintàctic. Les variables i els seus graus també els passem però amb la diferència que en aquest cas es guarda els seus valors doncs són d'utilitat per les fases posteriors. Els espais en blanc i comentaris que es troben a l'entrada els eliminem. Si no hi ha cap expressió regular que casi amb l'entrada aquesta serà considerada un error i es tractarà tal i com s'explica en l'apartat de **Tractament d'errors**.

4.3 Tractament dels errors lèxics

Els errors lèxics són els produïts en escriure malament un identificador, una paraula clau o un operador. O Dit d'un altra manera, quan NO es troba una expressió regular en la especificació JFLEX que concordi amb l'entrada que estem tractant.

El programa, quan detecta un error lèxic, l'indica i, sense aturar-se, en continua la execució. D'aquesta manera permet detectar tants errors com sigui possible.

El mecanismes de JFLEX per la gestió d'errors lèxics són primitius. Per tal d'augmentar l'eficiència de la seva gestió es passa els errors lèxics al analitzador sintàctic CUP. D'aquesta manera es gestionen com un error sintàctic amb els seus mecanismes de gestió més sofisticats.

4.4 Especificació lèxica del traductor

```
1
2  /********************** Codi Usuari
       ******************************/
3  import java_cup.runtime.*;
4  %%
5  /************************ Opcions
       *********************************/
6  %class Lexer
7  %unicode
8  %cup
9  %line
10 %{
11   private Symbol symbol(int type) {
12     return new Symbol(type, yyline, yycolumn);
13   }
14   private Symbol symbol(int type, Object value) {
15     return new Symbol(type, yyline, yycolumn, value);
16   }
17 %}
18 /********************** Delcaracions i Macros
       ************************/
19 FinalLinea = \r|\n|\r\n //EOF
20 Comentari = "/*" [^*] ~"*/" | "/*" "*"+ "/" | "//"+.*~\n
21 EspaiBlanc = {FinalLinea} | [ \t\f]
22 Digit = [0-9] Lletra = [A-Za-z]
23 Vble = {Lletra}({Lletra}|{Digit}|_)*
24 Grau = "0."{Digit}+| 1
25 %%
26 /********************************** Regles Lèxiques
       ******************/
27 <YYINITIAL> {
28     {EspaiBlanc} { /* ignorar */ }
29     {Comentari} { /* ignorar */ }
30
31         ";"     { return symbol(sym.SEMI); }
32     "("     { return symbol(sym.LPAREN); }
33     ")"     { return symbol(sym.RPAREN); }
34     "{"     { return symbol(sym.LCLAU); }
35     "}"     { return symbol(sym.RCLAU); }
36     ","     { return symbol(sym.COMMA); }
37     "<-"        { return symbol(sym.FLETXA); }
38         "^"             { return symbol(sym.AND); }
39         ""     { return symbol(sym.NEG); }
40     {Vble}  { return symbol(sym.VBLE, new String(yytext())); }
41     {Grau}  { return symbol(sym.GRAU, new Double(yytext())); }
42 }
43 /*Tractament d'errors*/
```

```
44  .|\n    { System.out.println("Error lexic"
45              + "caracter no valid <"+yytext()+">"); }
```

5 ANÀLISI SINTÀCTIC

5.1 Funció

És la segona etapa de la Fase d'Anàlisi d'un traductor. En l'anàlisi sintàctic es comprova la validesa de la seqüència de tokens que ens envia el analitzador lèxic en base a una gramàtica. Si l'entrada és vàlida, construeix una sèrie d'estructures que seran utilitzades en la següent fase del anàlisis, el semàntic.

5.1.1 Compil·lació dirigida per sintaxi

Tot i la distinció teòrica de les diferents fases d'un traductor/compil·lador aquestes no són independents les unes amb les altres sinó que és el analitzador sintàctic és el que dirigeix tot el procés de compil·lació. Totes les altres fases de la compil·lació estan subordinades a ell. Per aconseguir aquesta subordinació el analitzador sintàctic realitza aquestes altres funcions:

- Controla el flux de tokens que envia l'analitzador lèxic: cada cop que l'analitzador sintàctic necessita un token per construir les estructures de dades crida al analitzador lèxic que llavors llegeix uns quants caràcters de l'entrada fins a trobar un nou component que és el que envia al analitzador sintàctic. D'aquesta manera l'analitzador sintàctic va construint les estructures de dades fins que no hi ha més components d'entrada o fins que s'arriba a un error.
- Permet incorporar codi que es pot emprar per implementar la resta de parts del compilador exceptuant l'analitzador lèxic.
- Detecta errors sintàctics.

Es pot veure doncs la importància que té l'analitzador sintàctic en tot el procés.

5.2 Implementació

Al estar la compil·lació dirigida per sintaxi en l'analitzador sintàctic s'implementen part de les fases següents del compilador. Malgrat això, per motius de claredat, en aquest apartat només s'explica tot allò que fa referència al analitzador sintàctic. La resta de fases, tot i està incloses en el mateix arxiu, s'explicaran més endavant en els seus apartats corresponents.

El fitxer CUP del TFC consta dels següents apartats:

1. Especificacions *import* i *package*.
2. Codi d'usuari.
3. Llista de símbols.
4. Precedència i associativitat.
5. La gramàtica.

5.2.1 Especificacions import i package

En aquest apartat s'escriuen les declaracions *import* i *package* que tenen la mateixa sintaxi i funció que en un programa Java normal. Aquí s'escriu l'import del *java_cup.runtime* que ens permetrà emprar les eines de CUP.

5.2.2 Codi d'usuari

Després de les declaracions d'import i package es defineix el codi d'usuari. Aquest codi s'inclourà en les classes que formen part de l'analitzador sintàctic generat per CUP. El codi d'usuari es pot inserir en un lloc concret dins el codi del parser. Per això fem servir diferents declaracions que indiquen on s'insereix el codi. En parser code és on s'inclou el codi que fa referència al analitzador sintàctic en si i és el que s'explica a continuació. La resta, action code, s'explica en l'analitzador semàntic doncs forma part d'ell.

Dins de *parser code{: .... :};* s'inclou el codi, variables i mètodes, que van directament inserits en la classe del analitzador sintàctic. Dins de parser code també es poden sobreescriure mètodes que serveixen per personalitzar el comportament del parser. En el compil·lador es defineixen els següents mètodes:

- *public void report_error(String message, Object info)*: aquest mètode es crida sempre que hi ha un error sintàctic. La seva implementació per defecte és només treure per pantalla message. El mètode original s'ha sobreescrit perquè mostri també el lloc on es produeix aquest error.

- *protected int error_sync_size()*: aquest mètode es crida per determinar quants tokens s'han de llegir correctament per considerar que el parser es recupera d'un error satisfactòriament. Per defecte el seu valor és 3 i, segons el manual de CUP, no es recomana posar valors menors de 2. Però es posa el valor 1 per reduir al màxim els possibles errors no detectats.

- *public parser(java_cup.runtime.Scanner s, TaulaSimbols taula)*: és un nou constructor per la classe parser generada per CUP. Aquest constructor permet passar una taula que és l'estructura utilitzada per fer l'anàlisi semàntic i la generació de codi tal i com es veu en apartats posteriors.

- *public void obtenirToken()*: és un mètode que s'utilitza per guardar informació sobre el token que estem tractant actualment i que es fa servir en el analitzador semàntic per tenir informació d'on es produeix l'error.

5.2.3 Llista de símbols

En llista de símbol es dóna un nom i un tipus per a cada un dels terminals i no-terminals que apareixen en la especificació CUP del compil·lador. Els terminals són aquells símbols que son retornats per l'analitzador lèxic. Els símbols que no són retornats pel scanner es consideren no-terminals i han d'aparèixer almenys un cop en la part esquerra d'una regla de la gramàtica i serveixen per poder construir la gramàtica.

En la llista de símbols del compil·lador s'ha de ressaltar que només el terminals VBLE i GRAU tenen un valor que és del tipus String i Double respectivament. La resta de terminals i no terminals no tenen cap tipus de valor associat.

5.2.4 Precedència i associativitat

Per evitar les possibles ambigüitats en la gramàtica s'estableix, per a tots els terminals que actuen com operadors, una precedència i una forma d'associació. En el traductor/-compilador que es construeix hi han 3 operadors que segueixen les regles següents:

```
precedence nonassoc FLETXA;
precedence left AND;
precedence left NEG;
```

Les regles van ordenades de menys a més precedència, per tant, la darrera regla és la que té la precedència més gran. La primera regla ens indica que l'operador FLETXA (<-) no és associatiu, així doncs, dues ocurrències d'aquest operand en la mateixa producció serà considerat un error. Les dues últimes regles indiquen que tant AND (^) com NEG (¬) tenen associativitat per l'esquerra essent NEG (¬) qui té més precedència.

5.2.5 Gramàtica

En aquesta part del analitzador sintàctic es descriu com són les regles de producció de la gramàtica formal lliure de context que s'utilitza en el traductor. L'esquema de les regles en CUP és el següent:

no-terminal ::= β_1*{: Acció 1 :}*
|β_2*{: Acció 2 :}*
...
|β_n*{: Acció n :}*
;

Com es pot observar, cada producció de la gramàtica està formada per un no-terminal en la part esquerra seguit pel símbol "::=", el qual es seguit β_i que és un conjunt format per 0 o més accions, terminals o no terminals. Finalment s'acaba la regla amb el símbol ";". Si hi han múltiples produccions per al mateix no-terminal després de "::=" es posarà un "|" per separar cada producció i aquesta sempre acabarà amb ";".

Les accions de la regla són el codi Java que s'executa quan el parser ha reconegut la producció que hi ha a l'esquerra de l'acció. Aquest codi de l'acció està contingut entre els delimitadors *{: ... :}*.

En la part dreta d'un símbol, després de ":", es pot escriure, alternativament, una etiqueta que serveix per poder referir-nos al valor del símbol en la part d'accions de la regla. El nom d'aquesta etiqueta ha de ser únic en la producció.

En aquest apartat no es descriuen les accions que es realitzen en cada regla doncs són accions relacionades amb l'analitzador semàntic i la taula de símbols. Aquestes accions s'explicaran més endavant.

La gramàtica formal pel traductor ha d'acceptar només la entrada descrita en l'apartat 2. Per complir els seus requisits es fan servir les següents regles:

1. Estan formades per una o més produccions acabades en ";". Si no hi ha cap producció de *expr_list* que casi amb l'entrada llavors es declara un error sintàctic.

```
expr_list ::= expr_list linia SEMI
| linia SEMI
| error SEMI
;
```

2. Cada producció està formada per dues parts: variable i garantia.

```
linia ::= variable garantia
;
```

3. La part anomenada variable conté, entre parèntesis, un variable, negada o no, i el seu grau separats per una coma.

```
variable ::= LPAREN vble_dec COMMA GRAU:e RPAREN
;
vble_dec ::= VBLE:e
| NEG VBLE:e
;
```

4. La garantia està formada per una de les dues formes següents: un conjunt buit o una variable més el símbol d'implicació seguit per una o més variables separades pel símbol de la conjunció.

```
garantia ::= LCLAU RCLAU
|LCLAU vble FLETXA expr RCLAU
|LCLAU vble RCLAU
;
```

   on *expr* representa a una o més variables separades pel símbol de la conjunció a través de les següents produccions:

```
expr ::= pare
| pare expr_and
;
pare ::= VBLE:e
| NEG VBLE:e
;
expr_and ::= AND pare expr_and
| AND pare
;
```

5.3 Tractament dels errors sintàctics

Els errors sintàctics són els errors produïts quan hi ha un defecte en l'estructura o expressió d'una producció de l'entrada com poden ser parèntesis mal equilibrats o manca de variables. Concretament l'error es produeix quan l'analitzador sintàctic no pot realitzar ni un shift, reduce o accept; o dit d'un altra manera, no hi ha cap producció que casi amb l'entrada que s'està tractant. Quan es produeix aquesta situació llavors el parser genera un error sintàctic i entra en mode error.

Quan un error es detectat una porció de la pila és eliminada i reemplaçada pel símbol "error" de CUP. Si no hi hagués cap estat que pogués reduir per "error" llavors el parser avortaria. A l'hora d'escriure les produccions, el símbol "error" s'utilitza en la part dreta tal i com es mostra a continuació:

```
expr_list ::= expr_list linia SEMI
| linia SEMI
| error SEMI
;
```

Un cop l'analitzador sintàctic ha reduït pel símbol "error" descarta els següents tokens de l'input fins que troba el terminal ";" que es fa servir com a *token de sincronització*. Aquest token de sincronització serveix per evitar que el parser surti abans d'hora del mode d'error i serveix per descartar la sentència que s'estava llegint. Després del token de sincronització l'analitzador sintàctic ha de llegir satisfactòriament, sense executar cap acció semàntica, el número de tokens especificats al mètode *protected int error_sync_size()*. Si el parser llegeix correctament aquests tokens sortirà del mode error i continuarà llegint l'input de forma habitual executant totes les accions semàntiques. Si en canvi, no pot sortir del mode error perquè ha tornat a trobar un error el parser repetirà el procés anterior fins que surti del mode error o s'arribi al final de l'input.

La gestió dels errors que es fa servir en CUP d'ignorar els errors té una mancança. Al descartar una sèrie de tokens, tres per defecte, després del token de sincronització ";" si hi ha un error immediatament després de ";" aquest no serà detectat. Per tal de reduir al màxim aquest problema en l'apartat 5.2.2 s'explica que s'escull, mitjançant el mètode *protected int error_sync_size()*, que es llegeixi un sol token després del token de control.

Al descobrir un error sintàctic, com en els lèxics, l'indiquem i continuem l'anàlisi de l'entrada per detectar altres possibles errors sintàctics. Hem de tenir en compte que, al trobar un error, descartem el contingut de la producció fins que trobem un ";", per tant, és molt probable que un error sintàctic ens comporti més endavant un error semàntic que aturarà l'execució del programa tal i com veurem en el apartat 6.3.2 de tractament d'errors semàntics.

5.4 Especificació sintàctica del traductor

```
1 /*---Declaració dels Terminals i No Terminals ----------*/
2
```

```
3  terminal  SEMI, LPAREN, RPAREN, LCLAU, RCLAU, COMMA, FLETXA,
       AND, NEG;
4  terminal String   VBLE;
5  terminal Double   GRAU;
6  non terminal    expr_list, expr_and, vble_dec, linia, variable,
        garantia, expr, vble, pare;
7
8  /*---Precedència i Associativitat dels Terminals ---------*/
9
10 precedence nonassoc FLETXA;
11 precedence left AND;
12 precedence left NEG;
13
14 /* ----------------------Gràmatica--------------------- */
15
16 expr_list ::= expr_list linia SEMI
17         | linia SEMI
18         | error SEMI
19         ;
20 linia ::= variable garantia
21         ;
22 variable ::= LPAREN vble_dec COMMA GRAU:e RPAREN
23         ;
24 vble_dec ::= VBLE:e
25         | NEG VBLE:e
26         ;
27 garantia ::= LCLAU RCLAU
28         | LCLAU vble FLETXA expr RCLAU
29         | LCLAU vble RCLAU
30         ;
31 vble ::= VBLE:e
32         | NEG VBLE:e
33         ;
34 expr ::= pare
35         | pare expr_and
36         ;
37 pare ::= VBLE:e
38         | NEG VBLE:e
39         ;
40 expr_and ::= AND pare expr_and
41         | AND pare
42         ;
```

6 TRADUCCIÓ DIRIGIDA PER LA SINTAXI I LA TAULA DE SÍMBOLS

6.1 Visió general

Un cop fets els anàlisis lèxic i sintàctic el traductor passa a fer l'anàlisi semàntic. Aquesta fase s'encarrega de comprovar que l'entrada del traductor tingui un significat coherent. Això es fa principalment a partir de l'anàlisi dels tipus que apareixen a l'entrada així com la relació que tenen entre ells. Per exemple, l'anàlisi semàntic detectaria que una multiplicació entre una cadena de caràcters i un número, això és un error semàntic. Aquesta fase està íntimament lligada a la taula de símbols per això en aquest apartat es descriuen conjuntament.

6.2 Taula de símbols

La taula de símbols o taula d'identificadors és una estructura de dades d'alt rendiment que emmagatzema tota la informació necessària dels identificadors de l'usuari. Aquesta taula té dues funcions principals:

- Fer comprovacions semàntiques.
- Generar codi en fases posteriors del traductor.

Una adient i eficaç gestió de la taula de símbols és molt important perquè la seva manipulació consumeix gran part del temps de compil·lació o traducció.

6.2.1 Implementació de les variables del traductor

Tal i com s'ha dit, la taula de símbols guarda la informació de totes les variables que apareixeran en l'entrada del traductor. El primer pas és doncs definir quants tipus de variables possibles hi haurà en el traductor així com quina informació es guarda de cada variable.

El traductor només té un tipus de variable possible. D'aquest tipus de variable es guarda el seu nom, el valor del seu grau, si està negat i, si en té, els noms de tots els seus pares. Es defineix la classe *simbol* on es guarda tota aquesta informació:

```
1  class Simbol{
2    String nom;
3    Double grau;
4    Boolean negat;
5    Vector<String> pare;
6
7    public Simbol(){
8      this.nom = "buit";
9      this.grau = -1.0;
10     this.pare = new Vector<String>();
11     this.negat = false;
```

```
12      }
13  }
```

Els objectes de tipus símbol seran els elements que inserirem en la estructura de dades que es defineix a continuació.

6.2.2 Implementació de l'estructura de dades

Per poder fer l'anàlisi semàntic i la interpretació gràfica es fan servir tres tipus d'operacions en la taula de símbols:

- Inserir un element en la taula.
- Cercar un element dins la taula.
- Llistar tots els elements de la taula de forma ordenada.

La taula de símbols ha d'executar eficientment les operacions esmentades doncs són cabdals en la traducció. L'estructura de dades escollida és el *LinkedHashMap*[5] inclosa en l'API de Java. Aquesta estructura es comporta com una taula Hash però amb l'afegit que manté una Llista doblement enllaçada amb els elements seus elements ordenats, generalment, per ordre d'entrada. *LinkedHashMap* realitza les operacions d'inserció i cerca amb un cost constant amb un rendiment lleugerament inferior al d'una taula Hash degut a que manté la Llista enllaçada. A canvi d'aquest petit decrement de rendiment, l'estructura de dades permet llistar tots els elements de la taula de forma ordenada de manera molt més eficient que una taula Hash normal. Aquesta estructura de dades contindrà tots els objectes de tipus símbol que generarà el fitxer d'entrada. Tenint en compte l'esmentat fins ara la classe TaulaSimbols té la següent forma:

```
1   public class TaulaSimbols{
2
3           LinkedHashMap<String, Simbol> t;
4
5           public TaulaSimbols(){
6                   t = new LinkedHashMap<String, Simbol>();
7           }
8           public void inserir(String nom, Simbol s){
9                   t.put(nom, s);
10          }
11          public Simbol cercar(String nom){
12                  return (Simbol)(t.get(nom));
13          }
14          public boolean existeix(String nom){
15                  return t.containsKey(nom);
16          }
17          public Collection values(){
```

[5] http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html

```
18                  return t.values();
19          }
20          public int tamany(){
21                  return t.size();
22          }
23          public void imprimir(){
24                  Iterator it = t.values().iterator();
25                  while(it.hasNext()){
26                          Simbol sim = (Simbol)it.next();
27                          System.out.println(sim.nom +" : "+ sim.
                                negat+" : "
28                                  + sim.grau + "numero de pares "
29                                  + sim.pare.size() + "\n");
30                          Enumeration e = sim.pare.elements();
31                          while(e.hasMoreElements()){
32                                  System.out.print("Pare o Pare:
                                        "
33                                          + e.nextElement() +"\n"
                                            );
34                          }
35                  }
36          }
37  }
```

S'han utilitzat el mètodes *get*, *put*, *containsKey* i *iterator* de la classe *LinkedHashMap* per implementar les operacions que es necessiten per fer sobre la taula de símbols.

6.3 Anàlisi Semàntic

6.3.1 Interacció entre l'analitzador sintàctic, el semàntic i la taula de símbols

Un *atribut* és una informació que va associada a un terminal o no-terminal. Una *acció o regla semàntica* és codi que accedeix a la informació del atribut per executar una acció. Aquestes regles semàntiques s'associen amb les regles de producció del anàlisi sintàctic o, dit d'un altra manera, s'executa una acció semàntica al tractar una producció sempre hi quan no estiguem dins d'un error sintàctic. El motiu d'aquesta relació és una conseqüència de que, com ja hem dit anteriorment (Apartat 5.1.1), la compil·lació es dirigida per sintaxi.

Les accions semàntiques del present traductor tenen les següents funcions:

- Guarden la informació de l'entrada del traductor que després es necessitarà per poder fer la representació gràfica.
- Detecta errors semàntics i ens informa de quin tipus són.

Tant per guardar la informació de l'entrada com per detectar els errors semàntics el traductor fa servir la taula de símbols. Es pot veure doncs l'estreta relació que hi ha entre l'analitzador semàntic i la taula de símbols.

6.3.2 Tractament dels errors semàntics

Els errors semàntics són els errors produïts en utilitzar variables en àmbits que no li corresponen, dins l'entrada del programa, degut al seu valor.

Quan es troba un error de tipus semàntic, al contrari que en els de tipus lèxic i sintàctic, s'atura l'execució del programa. El motiu és que aquest tipus d'errors afecten a la coherència de la resta de l'entrada, i per tant, un error semàntic implica un error semàntic en tota l'entrada. No tindria sentit continuar l'execució doncs es podria arribar a detectar nous errors que en realitat no en serien. Els errors semàntics que detecta el programa són els següents:

- En la mateixa entrada no podem tenir una producció amb una variable P i un altra producció amb la seva negada $\neg P$.
- Com exigim una entrada ordenada, l'aparició d'una variable nova en la garantia d'una producció és un error semàntic.
- L'aparició de dues o més produccions que tinguin la mateixa variable a l'argument.
- Una garantia buida quan el grau de l'argument és < 1. Per ser considerat un fet la garantia ha de tenir la mateixa variable que en l'argument.
- Una garantia amb una regla quan el grau de l'argument és 1. Si accepta una garantia amb només la mateixa variable que hi ha a l'argument.
- La variable del argument i el conseqüent de la garantia són diferents. Exemple: $(Q, 0.7)\{R \leftarrow P\}$ és un error semàntic.
- L'antecedent o antecedents de la garantia tinguin un grau inferior al grau de l'argument. Exemple: en la producció $(Q, 0.7)\{Q \leftarrow P\}$ la variable P ha de tenir un grau superior o igual a 0.7.

El programa, a més de detectar els errors que acabem de descriure, ens indica la línia en que es produeix i el tipus d'error que és.

6.3.3 Implementació de l'analitzador semàntic

Action Code Dins del fitxer .cup que, posteriorment genera l'analitzador sintàctic, hi ha una secció anomenada *action code* que permet posar codi de l'usuari dins una classe no pública que genera el fitxer cup. S'ha utilitzat aquesta possibilitat per escriure dos mètodes que es faran servir en el tractament dels errors semàntics i que ajudaran a poder corregir aquests errors per part de l'usuari. Aquests mètodes són:

- *protected void error(int tipus_error)*: que serveix per mostrar quin tipus d'error semàntic s'ha produït en l'entrada. A més a més, aquest mètode aturarà l'execució del traductor.
- *protected void semantic_error(Object info)*: que mostra la línia i columna on es produeix un error semàntic.

Accions semàntiques Degut a que la compil·lació està dirigida per la sintaxi, les accions semàntiques estan associades a les regles de producció sintàctica. Es descriu a continuació quina o quines accions semàntiques es produeixen a cada producció sintàctica de la gramàtica. Les produccions que no tenen associat cap regla semàntica s'ometen. Les accions semàntiques són les següents:

- Si s'arriba a reduir per aquesta producció significa que no hi hagut cap error ni sintàctic ni semàntic en una línia del fitxer d'entrada. Per això l'acció semàntica que es produeix es inserir la variable que surt en la part de variable de la línia dins la taula de símbols.

```
linia ::=       variable garantia
        {:
                parser.taula.inserir (s.nom, s);
        :}
        ;
```

- L'acció semàntica que es produeix és emmagatzemar el valor "e", que indica el grau d'una variable, dins l'objecte símbol.

```
variable ::= LPAREN vble_dec COMMA GRAU:e RPAREN
                {:
                        s.grau = e;
                :}
                ;
```

- Tan si és una variable negada com si no, en la següent producció es mira si la variable que es llegeix està ja en la taula de símbols. Si la variable ja existeix significa que s'està produint un error semàntic doncs una variable està declarada més d'un cop en l'entrada. Si la variable no existeix creem un objecte símbol i s'agafa la informació del seu nom i de si és una variable negada o no.

```
vble_dec ::=    VBLE:e
                {:
                        if(parser.taula.existeix(e)){
                                error(3);
                        }else{
                                s = new Simbol();
                                s.nom = e;
                                s.negat = false;
                :}
                        |NEG VBLE:e
                {:
```

```
                        if(parser.taula.existeix(e)){
                                error(3);
                        }else{
                                s = new Simbol();
                                s.nom = e;
                                s.negat = true;
                :}
                ;
```

- Si hi ha una garantia buida significa que el seu grau és 1, a més a més, no s'accepta que hi hagi una variable amb grau 1 que aparegui escrita explícitament. En cas que això no es compleixi es generaran un error semàntic segons quin sigui el cas que s'incompleix.

```
garantia ::= LCLAU RCLAU
                {:
                        if(s.grau < 1){
                                error(4);
                        }
                :}
                |LCLAU vble FLETXA expr RCLAU
                {:
                        if(s.grau == 1){
                                error(5);
                        }
                :}
                |LCLAU vble RCLAU
                ;
```

- La variable que ha d'aparèixer en l'antecedent de la implicació en la part de la garantia de la producció ha de ser igual a la variable que apareix en la part de variable i no es pot permetre que aparegui una variable i la seva negada en una mateixa producció.

```
vble ::=        VBLE:e
        {:
                if(s.negat){
                        error(1);
                else(!s.nom.equals(e)){
                        error(6);
                }
        :}
        |                       NEG VBLE:e
        {:
                if(!s.negat){
```

```
12                          error(1);
13                  else(!s.nom.equals(e)){
14                          error(6);
15                  }
16          :}
17          ;
```

- Si almenys una de les variables de l'antecedent de la implicació que hi ha a la part de la garantia de la producció té grau més petit que el consequent llavors es produeix un error semàntic. També és un error si una d'aquestes variables de l'antecedent de la implicació no ha aparegut abans a l'entrada del traductor. Si l'entrada no genera cap d'aquests dos errors llavors les variables de l'antecedent s'afegeixen al vector on es s'emmagatzemen els noms dels símbols que són pares del símbol que s'està tractant en aquell moment.

```
1   pare ::=        VBLE:e
2           {:
3                   if(!parser.taula.existeix(e)){
4                           error(2);
5                   }else{
6                           Simbol aux;
7                           aux = parser.taula.cercar(e);
8                           if(aux.negat||(aux.grau < s.grau)){
9                           error(7);
10                          }else{
11                                  s.pare.add(e);
12                          }
13                  }
14          :}
15          |NEG VBLE:e
16          {:
17                  if(!parser.taula.existeix(e)||!s.negat){
                                                error(2);
18                  }else{
19                          Simbol aux;
20                          aux = parser.taula.cercar(e);
21                          if((!aux.negat)||(aux.grau < s.grau)){
22                          error(7);
23                          }else{
24                                  s.pare.add(e);
25                          }
26                  }
27          :}
28          ;
```

6.4 Implementació dirigida per la sintaxi del traductor

```
/* --------Preliminary Declarations Section----------*/
import java.io.*;
import java.util.*;
import java_cup.runtime.*;

action code {:
        Simbol s;
        protected void error(int tipus_error){
                switch(tipus_error){
                        case 1:
                                System.err.println("Error: en"
                                + " la mateixa entrada tenim"
                                + " una variable i la seva"
                                + " negada.");
                                break;
                        case 2:
                                System.err.println("Error: la"
                                + " entrada del programa no"
                                + " està ordenada o hi ha"
                                + " variable no declarada.");
                                break;
                        case 3:
                                System.err.println("Error: 2"
                                + " o més produccions amb la"
                                + " mateix variable a"
                                + " l'argument o tenim a la"
                                + " mateixa entrada una"
                                + " variable i la seva"
                                + " negada.");
                                break;
                        case 4:
                                System.err.println("Error:"
                                + "tenim un grau menor a "
                                + "1 a l'argument, falta "
                                + "la garantia.");
                                break;
                        case 5:
                                System.err.println("Error:"
                                + " tenim un grau 1 a"
                                + " l'argument, la garantia"
                                + " ha de ser buida.");
                                break;
                        case 6:
                                System.err.println("Error: "
                                + "No coincideixen les"
                                + " variables del argument i"
                                + "les de la garantia.");
```

```
48                                      break;
49                              case 7:
50                                      System.err.println("Error: "
51                                      + "l'antecedent/s de "
52                                      + "la garantia té un grau
                                            inferior al "
53                                      + "de l'argument.");
54                                      break;
55                              default:
56                                      System.out.println("Número"
57                                      + " d'error inexistent.");
58                                      break;
59                      }
60                      parser.obtenirToken();
61                      semantic_error(parser.token_actual);
62              }
63              protected void semantic_error(Object info){
64                      StringBuffer m = new StringBuffer("Error"
65                              + " Semàntic");
66                      if (info instanceof java_cup.runtime.Symbol) {
67                              java_cup.runtime.Symbol s =
68                                      ((java_cup.runtime.Symbol) info
                                            );
69                              if (s.left >= 0) {
70                                      m.append(" en la linia "+(s.
                                            left+1));
71                                      if (s.right >= 0)
72                                              m.append(", columna "+(
                                                    s.right+1));
73                              }
74                      }
75                      System.err.println(m);
76                      System.exit(1);
77              }
78      :};
79      //Codi posat directament a la classe parser
80      parser code {:
81              public TaulaSimbols taula = new TaulaSimbols();
82              public Symbol token_actual;
83              public void obtenirToken(){
84                      token_actual = cur_token;
85              }
86              public parser(java_cup.runtime.Scanner s, TaulaSimbols
                    taula){
87                      super(s);
88                      this.taula=taula;
89              }
90      /* Canviem el missatge d'error per defecte per incloure fila
91      i columna del error*/
92              public void report_error(String message, Object info) {
```

```
93                  StringBuffer m = new StringBuffer("Error"
94                          + "Sintactic");
95                  if (info instanceof java_cup.runtime.Symbol) {
96                          java_cup.runtime.Symbol s =
97                                  ((java_cup.runtime.Symbol) info
                                        );
98                          if (s.left >= 0) {
99                                  m.append(" en la linia "+(s.
                                        left+1));
100                                 if (s.right >= 0)
101                                         m.append(", columna "+(
                                                s.right+1));
102                         }
103                 }
104                 m.append(" : "+message);
105                 System.err.println(m);
106         }
107 /*Redefinim quants tokens hem de llegir abans de sortir
108 del estat error*/
109         protected int error_sync_size() {
110                 return 1;
111         }
112 :};
113 /*---Declaration of Terminals and Non Terminals Section---*/
114 terminal  SEMI, LPAREN, RPAREN, LCLAU, RCLAU, COMMA, FLETXA,
        AND, NEG;
115 terminal String   VBLE;
116 terminal Double   GRAU;
117 non terminal    expr_list, expr_and, vble_dec, linia, variable,
         garantia, expr, vble, pare;
118 /*---Precedence and Associatively of Terminals Section---*/
119 precedence nonassoc FLETXA;
120 precedence left AND;
121 precedence left NEG;
122 /* ----------------------Gràmatica--------------------- */
123 expr_list ::= expr_list linia SEMI
124         | linia SEMI
125         | error SEMI
126         ;
127 linia ::= variable garantia
128         {:
129                 parser.taula.inserir (s.nom, s);
130         :}
131         ;
132 variable ::= LPAREN vble_dec COMMA GRAU:e RPAREN
133         {:
134                 s.grau = e;
135         :}
136         ;
137 vble_dec ::= VBLE:e
```

```
138         {:
139                 if(parser.taula.existeix(e)){
140                         error(3);
141                 }else{
142                         s = new Simbol();
143                         s.nom = e;
144                         s.negat = false;
145                 }
146         :}
147         | NEG VBLE:e
148         {:
149                 if(parser.taula.existeix(e)){
150                         error(3);
151                 }else{
152                         s = new Simbol();
153                         s.nom = e;
154                         s.negat = true;
155                 }
156         :}
157         ;
158 garantia ::= LCLAU RCLAU
159         {:
160                 if(s.grau < 1){
161                 error(4);
162                 }
163         :}
164         | LCLAU vble FLETXA expr RCLAU
165         {:
166                 if(s.grau == 1){
167                         error(5);
168                 }
169         :}
170         | LCLAU vble RCLAU
171         ;
172 vble ::= VBLE:e
173         {:
174                 if(s.negat){
175                         error(1);
176                 }else if(!s.nom.equals(e)){
177                         error(6);
178                 }
179         :}
180         | NEG VBLE:e
181         {:
182                 if(!s.negat){
183                         error(1);
184                 }else if(!s.nom.equals(e)){
185                         error(6);
186                 }
187         :}
```

```
188         ;
189 expr ::= pare
190         | pare expr_and
191         ;
192 pare ::= VBLE:e
193         {:
194                 if(!parser.taula.existeix(e)){
195                         error(2);
196                 }else{
197                         Simbol aux;
198                         aux = parser.taula.cercar(e);
199                         if(aux.negat||(aux.grau < s.grau)){
200                                 error(7);
201                         }else {
202                                 s.pare.add(e);
203                         }
204                 }
205         :}
206         | NEG VBLE:e
207         {:
208                 if(!parser.taula.existeix(e)||!s.negat){
209                         error(2);
210                 }else{
211                         Simbol aux;
212                         aux = parser.taula.cercar(e);
213                         if((!aux.negat)||(aux.grau < s.grau)){
214                                 error(7);
215                         }else {
216                                 s.pare.add(e);
217                         }
218                 }
219         :}
220         ;
221 expr_and ::= AND pare expr_and
222         | AND pare
223         ;
```

7 REPRESENTACIÓ GRÀFICA

7.1 Introducció

La representació gràfica és la darrera part del compil·lador i per tant, ha d'estar relacionada amb la resta de fases. Concretament, la integració amb la resta del compil·lador s'aconsegueix mitjançant la taula de símbols. Aquesta estructura de dades, que es crea i s'omple en fases anteriors del compil·lador, es passa a la representació gràfica com un argument i així s'aconsegueix integrar totes les fases del compil·lador. La taula de símbols guarda, de forma ordenada, totes les variables de l'entrada així com la relació entre elles.

7.2 Selecció de l'algoritme de representació gràfica

Després d'un estudi previ dels principals algoritmes de representació gràfica, es decideix que no hi ha cap algoritme que permeti representar la sortida gràfica del compil·lador d'una manera precisa. Els principals problemes que s'observen al buscar una representació gràfica són:

- Poden existir nodes els quals no es relacionen amb cap altre node i, a més, aquests nodes poden aparèixer en qualsevol lloc de la representació gràfica.
- Els nodes estan agrupats en una sèrie de nivells.
- No es pot concloure que la majoria d'arestes tinguin una direcció comuna ja que les arestes poden anar entre vèrtex del mateix nivell o vèrtex de diferent nivell.
- Al augmentar el nombre de variables a representar augmenta proporcionalment la dificultat d'interpretació de la representació gràfica per part de l'usuari.

Malgrat les dificultats per escollir un algoritme de representació gràfica es considera que el algoritme de distribució jeràrquica o per capes s'acosta suficientment al resultat desitjat com per ser escollit el fonament sobre la qual es realitzarà la resta de la representació gràfica. El principal objectiu de la distribució jeràrquica és mostrar la direcció principal dins d'un graf dirigit al mateix temps que els nodes son distribuïts en una sèrie de capes ordenades per algun tipus de jerarquia.

Al algoritme de distribució jeràrquica se li fan les següents modificacions per tal de d'adaptar-lo a les nostres necessitats:

- Es decideix no executar un mètode que alinea els vèrtexs que formen part d'una mateixa capa o nivell de forma horitzontal. En lloc d'això, els vèrtex que formen part d'una mateixa capa es posicionen dins d'un pare en forma de quadrat tan de forma horitzontal com vertical. Al fer aquesta modificació s'aconsegueix una representació més clara del dibuix si hi ha vèrtexs sense cap aresta. Per aconseguir això es deixa d'executar el mètode *alignCells(String align, Object[] cells).*

- Es canvia l'estil d'orientació les arestes per un estil anomenat "entitat relació". Aquest estil uneix els vèrtex de forma més directa que els altres estils però, a canvi, les arestes tenen més tendència a sobreposar-se. Tot i les mancances es considera aquest estil el més clar de tots per aconseguir una bona representació gràfica. S'ha definit la variable *SYTLE_EDGE* com a *EDGESTYLE_ENTITY_RELATION* per tal d'aconseguir aquest efecte.

7.2.1 Mecanismes per facilitar la comprensió de la representació gràfica

A vegades la interpretació correcta de la representació gràfica es fa difícil per l'usuari. Els motiu pels quals es fa difícil la interpretació és que algunes arestes es sobreposen a les altres depenen el nombre de variables que hi ha i la relació que hi ha entre aquestes. Per tal de combatre aquest problema s'han implementat els següents mecanismes:

- Les capes o nivells de la representació gràfica, que estan formades per el conjunt de variables que tenen un mateix grau, es poden minimitzar per així poder visualitzar altres nivells amb més atenció. Això s'ha aconseguit mitjançant el mètode *public void setCellsLocked(boolean value).*
- Mitjançant el mètode *public void setCellsLocked(boolean value)* s'ha fet que tan els nodes com les capes que formen part de la representació gràfica es poden moure per tal de facilitar la comprensió del origen i el destí de les arestes.
- Les arestes tenen diferents colors per tal de diferenciar les arestes que relacionen vèrtexs del mateix nivell, que tenen color blau, de les arestes que relacionen vèrtex que estan en diferents nivells, que tenen color vermell.

7.2.2 Implementació de la representació gràfica

Idea general de la implementació Per fer la representació gràfica es llegeix la taula de símbols de forma ordenada des de l'inici. Al llegir la taula es faran dues accions principalment:

- Quan es troba la primera variable d'un nou grau es crea un vèrtex pare dins del qual s'inseriran totes les variables que tinguin el mateix grau que aquesta variable.
- De cada variable es llegirà el vector on es guarden els noms de les variables que són pares amb la variable. Per cada element del vector es crearà una aresta entre aquest element del vector i la variable que tractem.

A part de les accions anteriors es modifiquen les característiques del model del graf, de la distribució jeràrquica i de les arestes per aconseguir una representació més fidel del que es vol aconseguir tal i com s'explica en l'apartat anterior.

Codi font

```
1 /* Classe que construeix la representació gràfica del
     compil·lador dissenyat.
```

```
2   Es basa amb el Hierarchical layout per fer la representació. */
3   import javax.swing.JFrame;
4   import javax.swing.JPanel;
5   import javax.swing.*;
6   import com.mxgraph.swing.mxGraphComponent;
7   import com.mxgraph.view.mxGraph;
8   import com.mxgraph.view.mxEdgeStyle;
9   import com.mxgraph.view.mxStylesheet;
10  import com.mxgraph.io.mxCodec;
11  import com.mxgraph.layout.*;
12  import com.mxgraph.layout.hierarchical.*;
13  import com.mxgraph.model.*;
14  import com.mxgraph.model.mxCell;
15  import com.mxgraph.model.mxGeometry;
16  import com.mxgraph.util.mxRectangle;
17  import com.mxgraph.util.mxPoint;
18  import com.mxgraph.util.mxConstants;
19  import java.util.*;
20
21  public class Graph  {
22      private static final long serialVersionUID =
23          -2707712944901661771L;
24      private static TaulaSimbols taula = new TaulaSimbols();
25      public Graph(TaulaSimbols taula){
26          this.taula = taula;
27          mxCodec codec = new mxCodec();
28          mxGraphModel model = new mxGraphModel();
29          mxGraph graph = new mxGraph(model);
30          mxHierarchicalLayout layout = new
31              mxHierarchicalLayout(graph,
32              SwingConstants.WEST);
33          //Característiques del Layout
34          layout.setDisableEdgeStyle(false);
35          layout.setResizeParent(true);
36          layout.setMoveParent(true);
37          layout.setFineTuning(false);
38          layout.setParallelEdgeSpacing(40);
39          layout.setResizeParent(true);
40          graph.setCellsLocked(false);
41          graph.setEnabled(false);
42          graph.setConnectableEdges(false);
43          //Característiques de les Arestes
44          Map<String, Object> stil = new HashMap<String,
45              Object>();
46          stil.put(mxConstants.STYLE_ROUNDED, false);
47          stil.put(mxConstants.STYLE_EDGE,
48              mxConstants.EDGESTYLE_ENTITY_RELATION);
49          stil.put(mxConstants.STYLE_SHAPE,
50              mxConstants.SHAPE_CONNECTOR);
51          stil.put(mxConstants.STYLE_ENDARROW,
```

```
52                  mxConstants.ARROW_CLASSIC);
53              stil.put(mxConstants.STYLE_VERTICAL_ALIGN,
54                  mxConstants.ALIGN_TOP);
55              stil.put(mxConstants.STYLE_ALIGN,
56                  mxConstants.ALIGN_CENTER);
57              stil.put(mxConstants.STYLE_STROKECOLOR,
58                  "#6482B9");
59              stil.put(mxConstants.STYLE_FONTCOLOR,
60                  "#446299");
61              stil.put(mxConstants.STYLE_PORT_CONSTRAINT,
62                  mxConstants.DIRECTION_SOUTH);
63              stil.put(mxConstants.STYLE_ROUTING_CENTER_X,
64                  0.5);
65              stil.put(mxConstants.STYLE_ROUTING_CENTER_Y,
66                  0);
67              stil.put(mxConstants.STYLE_AUTOSIZE, 0);
68              mxStylesheet foo = new mxStylesheet();
69              foo.setDefaultEdgeStyle(stil);
70              graph.setStylesheet(foo);
71              Object parent = graph.getDefaultParent();
72              String vertex_style1 = new String("ROUNDED;"
73                  + align=align_left;"
74                  + verticalAlign=middle;"
75                  + "fontSize=16;fontStyle=3;"
76                  + strokeColor=black;"
77                  + fillColor=#DDDDDD;spacing=10;"
78                  + shape=swimlane");
79              String grau, nom;
80              Double grauaux = 0.0;
81              String m = new String();
82              String color = new String();
83              String nom_var = new String();
84              String nom_arest = new String();
85              Object nivells = new Object();
86              Object prova = new Object();
87              Object orig_arest = new Object();
88              Object dest_arest = new Object();
89              graph.getModel().beginUpdate();
90              try{
91                  Iterator it = taula.values().iterator();
92                  while(it.hasNext()){
93                      Simbol sim = (Simbol)it.next();
94                      grau = Double.toString(sim.grau);
95                      m = ("\t"+"     GRAU "+ grau);
96                      if(sim.negat.equals(true)){
97                          nom_var = ("" + sim.nom);
98                      }else{
99                          nom_var = (sim.nom);
100                     }
101                     //Tractament dels vertex al comencament dels
```

```
                    //nivells
                    if(!sim.grau.equals(grauaux)){
                        grauaux = sim.grau;
                        nivells = graph.insertVertex(parent, m, m,
                            0, 0, 0, 0, vertex_style1);
                        codec.putObject(grau, nivells);
                        prova = graph.insertVertex(codec.getObject
                            (grau), nom_var, nom_var, 0, 0, 50,
                            50, "fontStyle=1");
                        codec.putObject(sim.nom, prova);
                        Enumeration e = sim.pare.elements();
                        while(e.hasMoreElements()){
                            nom_arest = (String)e.nextElement();
                            orig_arest = codec.getObject(nom_arest)
                                ;
                            dest_arest = codec.getObject(sim.nom);
                            nom_arest = nom_arest + "-" + sim.nom;
                            if (model.getParent
                                (dest_arest).equals
                                (model.getParent(orig_arest))){
                                color = "strokeColor=#000099";
                            }else{
                                color = "strokeColor=#990000";
                            }
                            graph.insertEdge(codec.getObject(grau),
                                nom_arest, null, orig_arest,
                                dest_arest, color);
                        }
                    }else{//Tractament dels vertex adicionals
                        prova = graph.insertVertex(codec.getObject
                            (grau), nom_var, nom_var,
                            0, 0, 50, 50, "fontStyle=1");
                        codec.putObject(sim.nom, prova);
                        Enumeration e = sim.pare.elements();
                        while(e.hasMoreElements()){
                            nom_arest = (String)e.nextElement();
                            orig_arest = codec.getObject(nom_arest)
                                ;
                            dest_arest = codec.getObject(sim.nom);
                            nom_arest = nom_arest + "-" + sim.nom;
                            if (model.getParent
                               (dest_arest).equals(
                               model.getParent(orig_arest))){
                                color = "strokeColor=#000099";
                            }else{
                                color = "strokeColor=#990000";
                            }
                            graph.insertEdge(codec.getObject(grau),
                                nom_arest, null,
                                orig_arest, dest_arest, color);
```

```
                        }
                    }
                }
            }
            finally{
                graph.getModel().endUpdate();
            }
            graph.getModel().beginUpdate();
            try{
                layout.execute(parent);
            }
            finally
            {
                graph.getModel().endUpdate();
            }

            mxGraphComponent graphComponent = new
                mxGraphComponent(graph);
            JFrame frame = new JFrame("Representació Gràfica");
            frame.getContentPane().add(graphComponent);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(1000, 1000);
            frame.setVisible(true);
        }
    }
```

8 CONCLUSIONS

8.1 Descripció temporal de l'eina

Tot i que el desenvolupament del projecte es va estendre més de 3 anys per motius personals es pot especificar el temps invertit en cadascun dels apartats del projecte. Concretament els cost temporal, especificat per fases, és el següent:

- Comprensió general i disseny de l'esquema inicial del compil·lador : En aquesta fase s'entén, globalment, el que es demana als requisits i es dissenya l'esquema general del projecte. Això significa tenir que estudiar que és un compilador i les parts de les quals consta, així com, la seva interacció entre ells. Posteriorment es va passar a estudiar el cas concret que s'havia d'implementar i, per entendre'l millor i facilitar el treball posterior, es va dividir el projecte en diverses parts i es va especificar de què constava cada part. Finalment, en aquesta part, es van aprendre els rudiments de Java perquè era el llenguatge que es faria servir per implementar el compil·lador. El temps invertit en aquesta part va ser de 75 hores.

- Cerca de les eines adients per implementar el lexer i el parser: Durant la fase anterior es decideix que l'anàlisi lèxic i sintàctic estan fortament vinculats i que, a l'hora d'implementar, val la pena tractar-les simultàniament. Per tant, en aquesta fase, es cerquen les eines en Java que permeten crear aquestes parts i que, al mateix temps, facilitin una fàcil i ràpida interacció entre l'analitzador lèxic i sintàctic. S'escull les eines Jflex i Java Cup i s'aprèn com funcionen. Aquest part es realitza en 32 hores.

- Implementació del lexer i parser: En el transcurs d'aquesta part s'implementen, mitjançant les eines estudiades en la part anterior, els analitzadors lèxic i sintàctic. Aquest procés dura 24 hores.

- Disseny de l'anàlisi semàntic, taula de símbols i la representació gràfica: En aquest part primerament es van cercar i estudiar eines amb Java que permetin realitzar la representació gràfica del compil·lador. Un cop escollida l'eina, que és jgraphx, s'estudien les millors estructures de dades que permetin realitzar un anàlisis semàntic i, al mateix temps, permetin construir la representació gràfica d'una forma òptima. A continuació es fa una cerca dels algoritmes de representació gràfica que puguin ser més útils per la representació gràfica. Finalment, en aquesta fase es fan un seguit de proves, combinant tots els elements estudiats, per tal d'escollir la millor opció. La duració temporal és de 80 hores.

- Implementació de la traducció dirigida per sintaxi i la taula de símbols: S'implementa l'estructura de dades i l'anàlisi semàntic del compil·lador segons el disseny i idees creades en l'apartat anterior. El temps invertit és de 24 hores.

- Implementació de la sortida gràfica: té un cost temporal de 40 hores.

- Redactar la memòria: aquesta fase dura 64 hores.

En la següent taula es mostra el resum d'hores dedicades a cada part:

Part del projecte	Hores
Comprensió general i disseny de l'esquema inicial del compil·lador	75
Cerca de les eines adients per implementar el lexer i el parser	32
Implementació del lexer i parser	24
Disseny de l'anàlisi semàntic, taula de símbols i la representació gràfica	80
Implementació de la traducció dirigida per sintaxi i la taula de símbols	24
Implementació de la sortida gràfica del compil·lador	40
Redactar la memòria	64
Total d'hores del projecte:	**339**

Taula 1: Resum d'hores del projecte

8.2 Limitacions actuals de l'eina

Trivialment es pot deduir que la capacitat d'entendre, amb un sol cop de vista, qualsevol representació gràfica és inversament proporcional al nombre d'elements a representar. Aquesta relació directa entre complexitat de representació i nombre d'elements hauria de ser la única existent a l'hora de fer la representació gràfica del compil·lador. Malgrat això, la representació gràfica implementada no escala sempre proporcionalment amb el nombre d'elements. Depenent de l'entrada, concretament de la relació que hi hagi entre els diversos nodes, el traductor genera una representació gràfica que fa difícil la seva comprensió a primer cop d'ull i que obliga a l'usuari a moure els diferents nodes o grups de nodes per entendre correctament la informació. El motiu és que a vegades la relació entre diversos nodes fa que es sobreposin les arestes fent difícil saber el seu origen i destí.

Aquesta limitació crec que és la més important que presenta el traductor i és la part que seleccionaria per realitzar futures millores. Però, s'ha de tenir en compte que un canvi en l'apartat gràfic pot implicar un canvi en altres llocs ja que les parts d'un compil·lador estan íntimament lligades les unes a les altres.

8.3 Dificultats trobades

On rau, segons al meu entendre, la màxima dificultat en el disseny d'un traductor és en escollir la taula de símbols i la seva relació amb l'anàlisi semàntic i la representació gràfica. Una estructura de dades pot ser ideal per realitzar les funcions d'anàlisis semàntic i, en canvi, ser poc adient per poder fer la representació gràfica. També pot passar el cas invers. Al dissenyar aquesta part s'ha de buscar el equilibri intentant trobar una solució que permeti fer un anàlisi semàntic òptim al mateix temps que no es perjudica la fase de construcció de la representació gràfica. Un canvi en aquest apartat afecta totes les altres. Les possibles solucions són múltiples i diverses i crec que és aquí on rau el moll de l'os del disseny de compil·ladors.

8.4 Tecnologies i eines utilitzades com a creixement personal

Per tal de ser capaç de realitzar el projecte he hagut de formar-me en eines que eren noves per a mi. Concretament he après les següents tecnologies:

- Aprendre Java per a poder realitzar el compilador en aquest llenguatge.
- Estudiar el que és un compil·lador, quin és el seu funcionament, les parts de les quals consta i quina relació hi ha entre les seves parts. Un cop assolits aquests conceptes, he tingut que estudiar com realitzar l'anàlisi lèxic i sintàctic amb les eines jflex i java cup respectivament.
- Estudiar algoritmes i conceptes de representació gràfica de grafs.
- He hagut d'estudiar i buscar diferents estructures de dades, algunes diferents a les estudiades durant la carrera, per tal de trobar l'estructura més adient per realitzar la taula de símbols.
- Per últim he tingut d'estudir l'eina de creació de grafs amb Java jgraphx.

L'amplitud de coneixements adquirits durant la realització del projecte del final de carrera han servit per augmentar els meus coneixements tècnics en molts camps. He trobat molt útil estudiar que és i com funciona un compil·lador doncs m'ha ajudat a entendre el perquè d'algunes característiques dels llenguatges de programació. Però, la part que he trobat més interessant és la complexitat que té el disseny d'un compil·lador sobretot la relació entre la taula de símbol, l'anàlisi semàntic i la representació gràfica. Observar les múltiples i complexes possibilitats de disseny que tenen les tres parts esmentades m'ha fet adonar de la gran dificultat que té realitzar un bon compil·lador. Per acabar també he trobat força interessant la complexitat que té modificar o crear algoritmes de visualització de grafs.

9 BIBLIOGRAFIA

9.1 Llibres

- Gálvez Rojas, Sergio; Mora Mata, Miguel Ángel. *Compiladores. Traductores y compiladores con lex/yacc, jflex/Cup y JavaCC*. 1ª ed. Málaga. Universidad de Málaga, 2005.
- Watt, David; Brown, Deryck. *Programming Language Processors in Java: Compilers and Interpreters*. 1ª ed. Prentice Hall, 2000.

9.2 Pàgines Web

- Berk, Elliot; Rowe, Steve; Décamps, Régis. *JFlex User's Manual*. Versió 1.4, 2013. Disponible a <http://www.jflex.de/manual.html>
- Petter, Michael; Hudson, Scott. *CUP User's Manual*. Versió 0.11b. Disponible a <http://www2.cs.tum.edu/projects/cup/docs.php>
- JGraph Ltd. *JGraphX (JGraph 6) User Manual*. Versió 2.8.1.1. Disponible a <http://jgraph.github.io/mxgraph/docs/manual_javavis.html>

9.3 Altres fonts

- Alsinet, Teresa. Processadors de Llenguatge: Apunts de Laboratori. Ciències de la Computació i Intel·ligència Artificial. Departament d'Informàtica i Enginyeria Industrial. Universitat de Lleida, Curs 2003/4.